Page-Differential Logging: An Efficient and DBMS-independent Approach for Storing Data into Flash Memory

Yi-Reun Kim*, Kyu-Young Whang*, Il-Yeol Song**

*Department of Computer Science

Korea Advanced Institute of Science and Technology (KAIST)

**College of Information Science and Technology

Drexel University

e-mail: *{yrkim, kywhang}@mozart.kaist.ac.kr, **songiy@drexel.edu

Abstract

Flash memory is widely used as the secondary storage in lightweight computing devices due to its outstanding advantages over magnetic disks. Flash memory has many access characteristics different from those of magnetic disks, and how to take advantage of them is becoming an important research issue. There are two existing approaches to storing data into flash memory: page-based and log-based. The former has good performance for read operations, but poor performance for write operations. In contrast, the latter has good performance for write operations when updates are light, but poor performance for read operations. In this paper, we propose a new method of storing data, called *page-differential logging*, for flash-based storage systems that solves the drawbacks of the two methods. The primary characteristics of our method are: (1) writing only the difference (which we define as the *page-differential*) between the original page in flash memory and the up-to-date page in memory; (2) computing and writing the page-differential only once at the time the page needs to be reflected into flash memory. The former contrasts with existing page-based methods that write the whole page including both changed and unchanged parts of data or from log-based ones that keep track of the history of all the changes in a page. Our method allows existing disk-based DBMSs to be reused as flash-based DBMSs just by modifying the flash memory driver, i.e., it is DBMS-independent. Experimental results show that the proposed method is superior in I/O performance, except for some special cases, to existing ones. Specifically, it improves the performance of various mixes of read-only and update operations by 0.5 (the special case when all transactions are read-only on updated pages) $\sim$ 3.4 times over the page-based method and by 1.6 $\sim$ 3.1 times over the log-based one for synthetic data of approximately 1 Gbytes. The TPC-C benchmark also shows improvement of the I/O time over existing methods by 1.2 $\sim$ 6.1 times. This result indicates the effectiveness of our method under (semi) real workloads.

1 Introduction

Flash memory is a non-volatile secondary storage that is electrically erasable and reprogrammable [4, 10]. Flash memory has outstanding advantages over magnetic disks: lighter weight, smaller size, better shock resistance, lower power consumption, and faster access time [10, 14, 25]. Due to these advantages, the flash memory is widely used in embedded systems and mobile devices such as mobile phones, MP3 players, and digital cameras [14, 15].

Flash memory is much different from a magnetic disk in structures and access characteristics [12]. It is composed of a number of blocks, and each block is composed of a fixed number of pages. It does not have seek and rotation latency because it is made of electronic circuits without mechanically moving parts [12]. Flash memory provides three kinds of operations — read, write, and erase. In order to overwrite existing data in a page, an erase operation must be performed before writing new data on the page [12, 14]. The write and erase operations are much slower than the read operation [14, 18]. Besides, the unit of the erase operation is a block, while the unit of the read and write operations is a page [25].

There have been a number of studies [2, 3, 8, 13, 14, 21] on the method of storing updated pages into flash memory for flash-based storage systems. In this paper, we refer to such methods as *page update* methods. The page update methods are classified into two categories [25] — page-based [3, 13] and log-based [2, 14, 21]. Page-based methods write the whole page into flash memory when an updated page *needs to be reflected* into flash memory (e.g., when the page is swapped out from the DBMS buffer to the database) [3, 13, 25]. These methods actually read only one page when *recreating* a page from flash memory (e.g., reading it into a DBMS buffer). Thus, they have good read performance. However, they have relatively poor write performance because they write the whole page including unchanged parts as well as changed parts of data [25]. In order to overcome this drawback, log-based methods have been proposed [25]. These methods write only the changes (which we call an *update log*[1]) in the page into the write buffer, which in turn is written into flash memory when the buffer is full [2, 14, 21]. Thus,

[1] An update log contains the changes in a page resulted in a single update command.

compared with page-based methods, log-based ones have good write performance when updates are not heavy [2] [25]. Log-based methods, however, have relatively poor read performance because they keep the history of all the changes (i.e., multiple update logs) in a page. Whenever an update is done, they write an update log into the write buffer. Thus, when updates are done multiple times, the update logs are likely to be written into multiple pages in flash memory. Thus, log-based methods need to read multiple pages when recreating a page from flash memory.

In this paper, we propose a page update method called *page-differential logging* (*PDL*) for flash-based storage systems. A *page-differential* (simply, a *differential*) is defined as the difference between the original page in the flash memory and the up-to-date page in memory. This novel method is much different from page-based methods or log-based ones in the following ways. (1) We write only the differential of an updated page. This characteristic stands in contrast with page-based methods that write the whole page including changed and unchanged parts of data or log-based ones that keep track of the history of all the changes (i.e., multiple update logs) in a page. Furthermore, we compute and write the differential only once at the time the updated page needs to be reflected into flash memory. The overhead of generating the differential is relatively minor because, in flash memory, the speed of read operation is much faster than those of write or erase operations. (2) When recreating a page from flash memory, we need fewer read operations than log-based ones do because we read at most two pages: the original page and the single page containing the differential. (3) When we need to reflect an updated page into flash memory, we need fewer write operations than others do because we write only the differential. A side benefit is that the longevity of flash memory is also improved due to fewer erase operations resulted from fewer write operations. (4) Our method is loosely-coupled with the storage system while the log-based ones are tightly-coupled. The log-based methods need to modify the storage management module of the DBMS because they must identify the changes in a page whenever it is updated. These changes can be identified only inside the storage management module because they are internally maintained by the system. On the other hand, our method does not need to modify the module of the DBMS because it computes the differential outside the storage management module by

[2] When pages are frequently updated, the log-based methods could be poorer in performance as we see in the experiments in p. 30, Figure 13.

comparing the page that needs to be reflected with the original page in the flash memory. We elaborate on this point later in Section 4.

The contributions of this paper are as follows. (1) we propose a new notion of "differential" of a page. Using this notion, we then propose a new approach to updating pages that we call *page-differential logging*. (2) Our method is DBMS-independent. (3) Through extensive experiments, we show that the overall read and write performance of our method is mostly superior to those of existing ones.

Hereafter, in order to reduce ambiguity in this paper, we distinguish logical pages from physical pages. We call the pages in memory *logical pages* and the ones in flash memory *physical pages*. For ease of exposition, we assume that the size of a logical page is equal to that of a physical page.

The rest of this paper is organized as follows. Section 2 introduces flash memory. Section 3 describes prior work related to the page update methods for flash-based storage systems. Section 4 presents a new page update method called *page-differential logging*. Section 5 presents the results of performance evaluation. Section 6 summarizes and concludes the paper.

2 Flash Memory

Based on the structure of memory cells, there are two major types of flash memory [6]: the NAND type and the NOR type. The former is suitable for storing data, and the latter for storing code [16]. In the rest of this paper, we use the term 'flash memory' to indicate the NAND type flash memory, which is widely used in flash-based storage systems [3].

Figure 1 shows the structure of flash memory. The flash memory consists of N_{block} *blocks*, and each block consists of N_{page} *pages*. A page is the smallest unit of reading and writing data, and a block is the smallest unit of erasing data [25]. Each page consists of a data area used for storing data and a spare area used for storing auxiliary information such as the valid bit, obsolete bit, bad block identification, and error correction check (ECC) [16].

[3] In this paper, we focus on flash memory but not on solid state disks (*SSD's*) [19], which have controllers with their own page update methods.

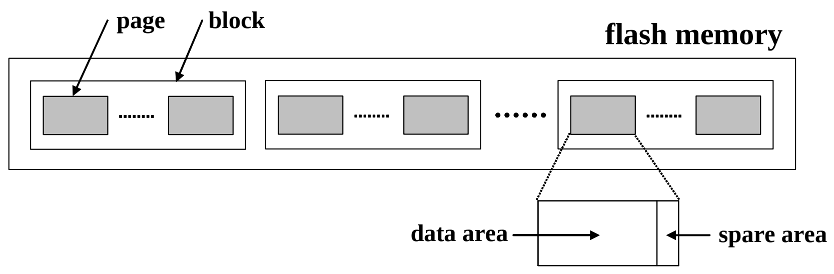


Figure 1. The structure of flash memory.

We consider three operations: read, write, and erase [6].

- **The read operation:** returns all the bits in the addressed page
- **The write operation:** changes a set of bits selected in the target page from 1 to 0
- **The erase operation:** sets all the bits in the addressed block to 1

The operations in flash memory are different from those in the magnetic disk in two ways. First, all the bits in flash memory are initially set to 1. Thus, writing to flash memory means selectively changing some bits in a page from 1 to 0. Next, the erase operation in flash memory changes the bits in a block back to 1. Each block can sustain only a limited number of erase operations before becoming unreliable, which is restricted to about 100,000 [4] [14, 15].

Due to the restriction of the write and erase operations, a write operation is usually preceded by an erase operation in order to overwrite a page [12, 14]. We first change all the bits in the block to 1 using an erase operation, and then, change some bits in the page to 0 using a write operation. We note that the erase operation is performed in a much larger unit than a write operation, i.e., the former is performed on a block while the latter on a page. The specific techniques for overwriting a page depend on the page update method employed. These techniques are discussed in Section 3.

Based on the capacity of memory cells, there are two types of flash memory [12]: Single Level Cell (SLC)-type and Multi Level Cell (MLC)-type. The former is capable of storing one data bit per

[4] Due to this characteristic, there have been a number of studies on wear-leveling [10] and bad block management [16]. However, we do not address them in this paper, but these studies can be applied to the storage system independently of the page update methods discussed in this paper.

cell, while the latter is capable of storing two (or even more) data bits per cell. Thus, MLC-type flash memory has greater capacity than SLC-type one and is expected to be widely used in high-capacity flash storages [12]. Table 1 summarizes the parameters and values of MLC flash memory we use in our experiments. We note that the size of a page is 2,048 bytes, and a block has 64 pages. In addition, the access time of operations increases in the following order: read, write, and erase. The read operation is 9.2 times faster than the write operation, which is 1.5 times faster than the erase operation.

Table 1. The parameters and values of flash memory*.

Symbols	Definitions	Values
N_{block}	the number of blocks	$32,768$
N_{page}	the number of pages in a block	64
S_{block}	the size of a block (bytes) (= $N_{page} \times S_{page}$)	$135,168$ ($64 \times 2,112$)
S_{page}	the size of a page (bytes) (= $S_{data} + S_{spare}$)	$2,112$ (= $2,048 + 64$)
S_{data}	the size of data area in a page (bytes)	$2,048$
S_{spare}	the size of spare area in a page (bytes)	64
T_{read}	the read time for a page (μs)	110
T_{write}	the write time for a page (μs)	1010
T_{erase}	the erase time for a block (μs)	1500

* Samsung K9L8G08U0M 2 Gbytes MLC NAND flash memory [18]

3 Related Work

The Page-Based Approach

In page-based methods [3, 13], a logical page is stored into a physical page. When an updated logical page needs to be reflected into flash memory, the whole logical page is written into a physical page [25]. When a logical page is recreated from flash memory, it is read directly from a physical page. These methods are loosely-coupled with the storage system because they can be implemented in a middle layer, called the *Flash Translation Layer* (*FTL*) [3], which maintains logical-to-physical address mapping between logical and physical pages as shown in Figure 2. The FTL can be implemented as hardware in the controller residing in SSD's, or can be implemented as software in the operating system for embedded boards [5].

[5] Commercial FTL's for SSD's or embedded boards typically use page-based methods [1]

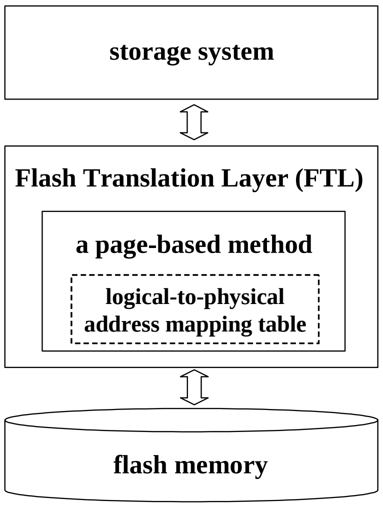
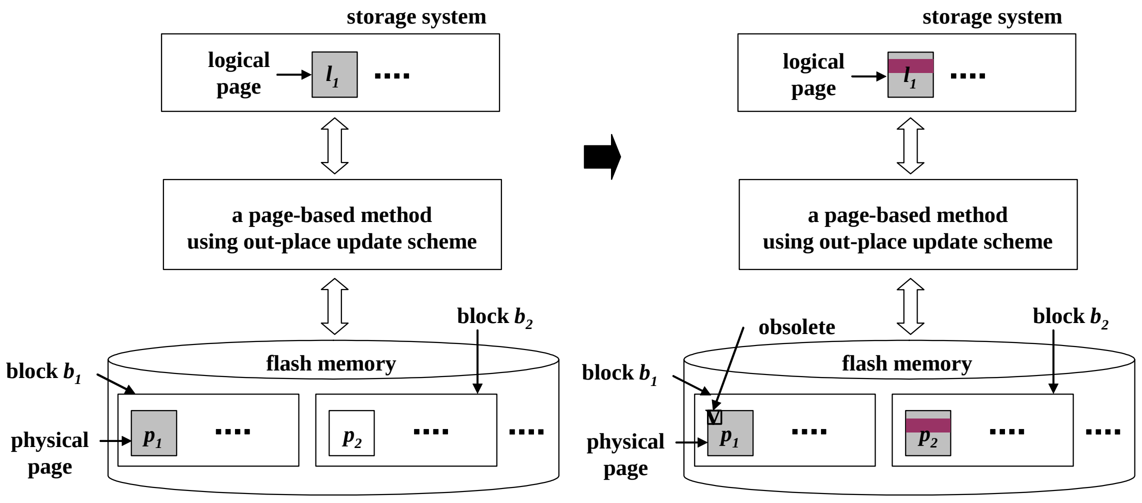


Figure 2. The architecture of the page-based method.

In page-based methods, there are two update schemes [15] — in-place update and out-place update — depending on whether or not the logical page is always written into the same physical page. When a logical page needs to be reflected into flash memory, the in-place update overwrites it into the specific physical page that was read [15], but the out-place update writes it into a new physical page [4, 25].

In-Place Update: As explained in Section 2, the write operation in flash memory cannot change bits in a page to 1. Therefore, when overwriting the logical page l_1 that was read from the physical page p_1 in the block b_1 into the same physical page p_1, we do the following four steps: (1) read all the pages in b_1 except p_1; (2) erase b_1; (3) write l_1 into p_1; (4) write all the pages read in Step (1) except l_1 in the corresponding pages in b_1. The in-place update scheme suffers from severe performance problems and is rarely used in flash memory [15] because it causes an erase operation and multiple read and write operations whenever we need to reflect a logical page into flash memory.

Out-Place Update: Figure 3 shows a typical example of the out-place update scheme. Figure 3 (a) shows the logical page l_1 read from the physical page p_1 in the block b_1. Figure 3 (b) shows the updated logical page l_1 and the two physical pages p_1 and p_2 — the original page read and the new page written. In order to overcome the drawback of in-place update, when we need to reflect the logical page l_1 into flash memory, the out-place update scheme first writes l_1 into a new physical page p_2, and then, sets p_1 to obsolete[6]. When there is no more free page in flash memory, a block is selected and obsolete pages in it are reclaimed by garbage collection [6], which converts obsolete pages to free pages. The out-place update scheme is widely used in flash-based storage systems [25] because it does not cause an erase operation when a logical page is to be reflected into flash memory.



(a) The logical page l_1 read from the physical page p_1. (b) The updated logical page l_1 and the process of writing it into the physical page p_2.

Figure 3. An example of out-place update.

The Log-Based Approach

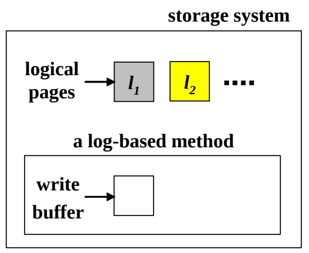
In log-based methods [2, 14, 21], a logical page is generally stored into multiple physical pages [14]. Whenever logical pages are updated, the update logs of multiple logical pages are first collected into a write buffer in memory [25]. When this buffer is full, it is written into a single physical page. Thus, when a logical page is updated many times, its update logs can be stored into multiple physical pages. Accordingly, when recreating a single logical page, multiple physical pages may need to be read and

[6]We set a page to obsolete by changing the obsolete bit in the spare area of the page from 1 to 0 as in Gal et al. [6].

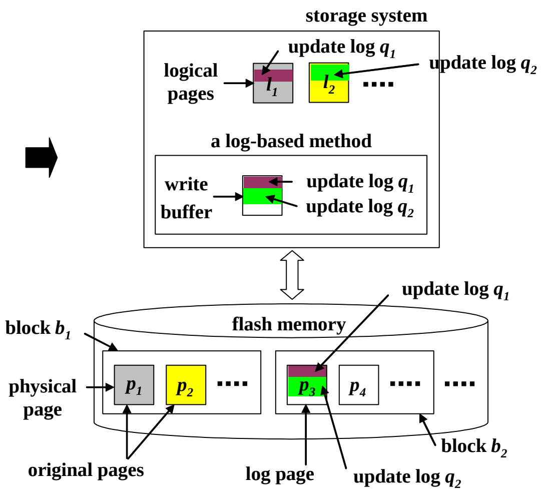
merged. The log-based methods are tightly-coupled with the storage system because the storage system must be modified to be able to identify the update logs of a logical page.

Among log-based methods, there are *Log-structured File system* (*LFS*) [17], *Journaling Flash File System* (*JFFS*) [21], *Yet Another Flash File System* (*YAFFS*) [2], and *In-Page Logging* (*IPL*) [14]. In LFS, JFFS, and YAFFS, the update logs of a logical page can be written into arbitrary log pages in flash memory while, in IPL, the update logs should be written into specific log pages. IPL divides the pages in each block into a fixed number of *original pages* and *log pages*. It writes the update logs of a logical page into only the log pages in the block containing the original (physical) page of the logical page. Therefore, when recreating the logical page, IPL reads the original page and only the log pages in the same block. When there is no free log page in the block, IPL merges the original pages with the log pages in the block, and then, writes the merged pages into pages in a new block (this process is called *merging* [14]). The old block is subsequently erased and garbage-collected. Consequently, IPL improves read performance by reducing the number of log pages to read from flash memory when recreating a logical page because log pages do not increase indefinitely (i.e., is bound) due to merging. The performance of IPL is similar to other log-based methods since IPL inherits the advantages and drawbacks of log-based methods other than the effect of merging and bound read performance.

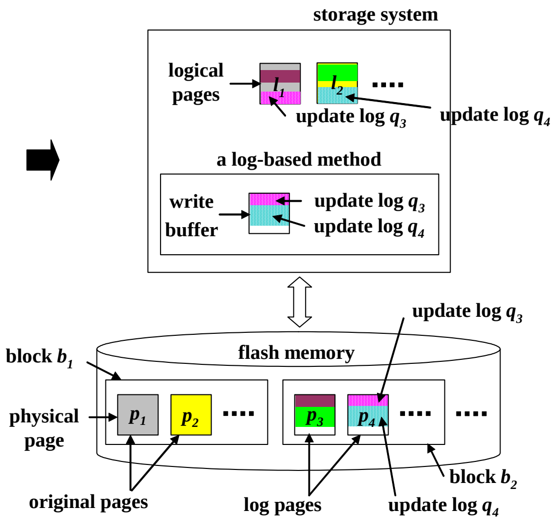
Figure 4 shows a typical example of the log-based methods. Figure 4 (a) shows the logical pages l_1 and l_2 in memory. Figure 4 (b) shows the update logs q_1 and q_2 of logical pages l_1 and l_2, respectively, and the process of writing them into flash memory. Here, the update logs q_1 and q_2 are first written into the write buffer, and then, the content of the write buffer is written into the log page p_3. Thus, the update logs q_1 and q_2 are collected into the same log page p_3. Figure 4 (c) shows a similar situation for the update logs q_3 and q_4 of logical pages l_1 and l_2. Figure 4 (d) shows the logical page l_1 being recreated from flash memory. Here, l_1 is recreated by merging the original page p_1 with the update logs q_1 and q_3 read from the log pages p_3 and p_4, respectively.



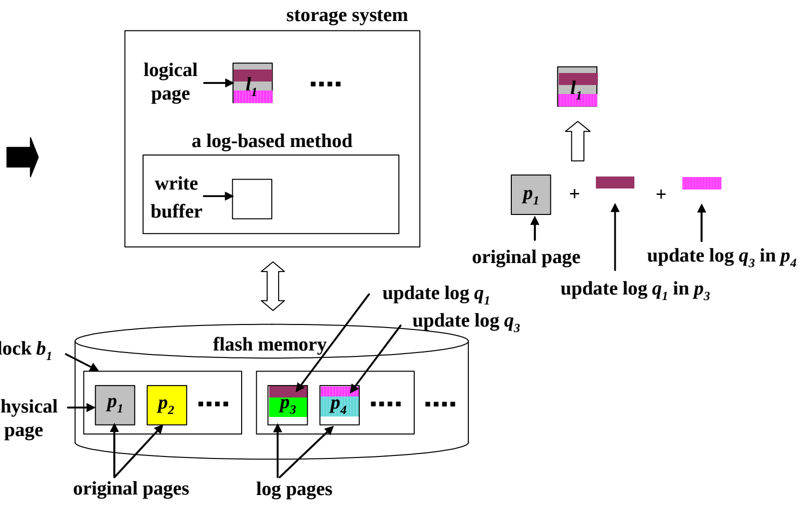
(a) The logical pages l_1 and l_2 in memory.



(b) The update logs q_1 and q_2 of logical pages l_1 and l_2, and the process of writing them into the log page p_3 in flash memory.



(c) The update logs q_3 and q_4 of logical pages l_1 and l_2, and the process of writing them into the log page p_4 in flash memory.



(d) The logical page l_1 being recreated from flash memory.

Figure 4. An example of the log-based approach.

4 The Page-Differential Logging Approach

In this section, we propose *page-differential logging* (*PDL*) for flash-based storage systems. Section 4.1 explains the design principles, and then, presents PDL, which conforms to these principles. Section 4.2 and 4.3 present the data structures and algorithms. Section 4.4 discusses the strengths and limitations.

4.1 Design Principles

We identify three design principles for PDL in order to guarantee good performance for both read and write operations. These principles overcome the drawbacks of both the page-based methods and the log-based methods in the following ways.

- **writing difference only :** We write only the *difference* when a logical page needs to be reflected into flash memory.
- **at-most-one-page writing :** We write *at most one* physical page when a logical page needs to be reflected into flash memory even if the page has been updated in memory multiple times.
- **at-most-two-page reading :** We read *at most two* physical pages when recreating a logical page from flash memory.

Page-differential logging method conforms to these three design principles. In this method, a logical page is stored into two physical pages — a *base page* and a *differential page*. Here, the base page contains a whole logical page, which could be the old version, and the differential page contains the difference between the base page and the up-to-date logical page. A differential page can contain differentials of multiple logical pages. Thus, the differentials of two logical pages could be stored in the same differential page.

The differential has the following advantages over the list of update logs in the log-based methods. (1) It can be computed without maintaining all the update logs, i.e., it can be computed by comparing the updated logical page with its base page only when the updated logical page needs to be reflected into flash memory. (2) It contains only the difference from the original page for the part that has been updated multiple times in a logical page. When a specific part in a logical page is updated in memory multiple times, the list of update logs contains all the history of changes while the differential contains only the difference between original data and the up-to-date data. For instance, let us assume that a logical page is updated in memory twice as follows: $\ldots aaaaaa \ldots \rightarrow \ldots bbbbba \ldots \rightarrow \ldots bcccba \ldots$. Here, the list of update logs contains two changes *bbbbb* and *ccc* while the differential contains only the difference *bcccb*.

In PDL, when an updated logical page needs to be reflected into flash memory, we create a differential by comparing the logical page with the base page in flash memory, and then, write the differential into the one-page write buffer, which is subsequently written into flash memory when it is full. Therefore, it conforms to the **writing-difference-only** principle.

We note that, when a logical page is simply updated, we just update the logical page in memory without recording the log. Instead, we defer creating and writing the differential until the updated logical page needs to be reflected into flash memory. Thus, our method satisfies the **at-most-one-page writing** principle.

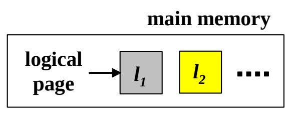
Theoretically, the size of the differential cannot be larger than that of one page. However, practically, it could be larger if a large part of the page has been updated. This case can occur since the differential contains not only the changed data but also the meta data such as offsets and lengths. In this case, we discard the created differential and write the updated logical page itself into flash memory as a new base page in order to satisfy the **at-most-one-page writing** principle. (In this special case, PDL becomes the same as the page-based method.)

When recreating a logical page from flash memory, we read the base page and its corresponding differential page, and then, merge the base page with its differential in the differential page. However, we need to read only one physical page if the base page has not been updated (i.e., there is no differential page). Thus, we need to read at most two physical pages, and accordingly, PDL conforms to the **at-most-two-page reading** principle.

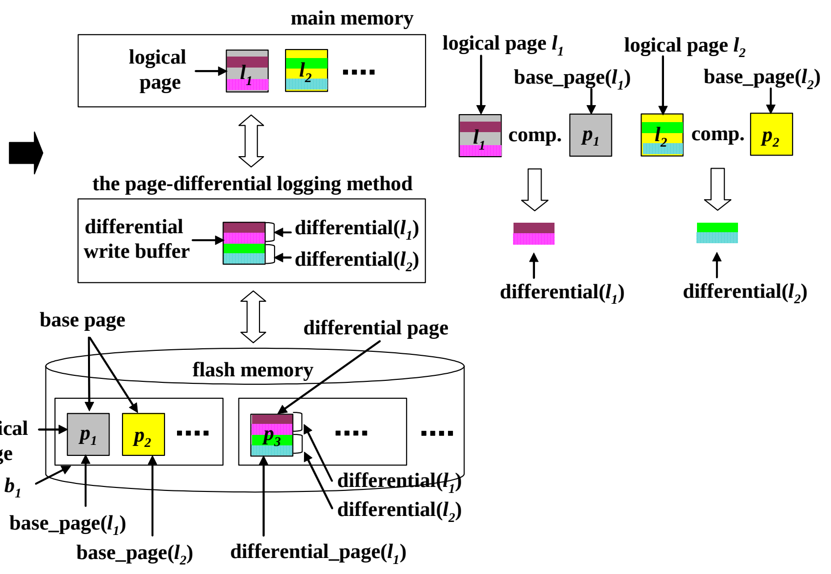
When there is no more free page in flash memory, obsolete pages are reclaimed by garbage collection. Here, we select one block for garbage collection. Since it may contain valid base or differential pages, before erasing the block, we move those valid pages into a new block, which is reserved for the garbage collection process [6]. For differential pages, however, we move only valid differentials into a new differential page, i.e., we do compaction here. Our method requires fewer write operations than page-based or log-based ones do because it satisfies the writing-difference-only and at-most-one-page writing principles. Thus, our method invokes garbage collection less frequently than other methods do.

Figure 5 shows an example of PDL. Here, we have base_page(p), differential_page(p), and differential(p) for the logical page p. Figure 5 (a) shows the logical pages l_1 and l_2 in memory. Figure 5 (b) shows the updated logical pages l_1 and l_2, and the process of writing them into flash memory. When l_1 and l_2 need to be reflected into flash memory, we perform the following three steps: (1) read the base pages p_1 and p_2 from flash memory; (2) create differential(l_1) and differential(l_2) by comparing l_1 and l_2 with the base pages p_1 and p_2, respectively; (3) write differential(l_1) and differential(l_2) into the write buffer, which is subsequently written into the physical page p_3 when the buffer is full. We note that l_1 and l_2 from different logical pages are written into the same differential page p_3. Figure 5 (c) shows the logical page l_1 recreated from flash memory by merging the base page p_1 with differential(l_1) in p_3 [7].

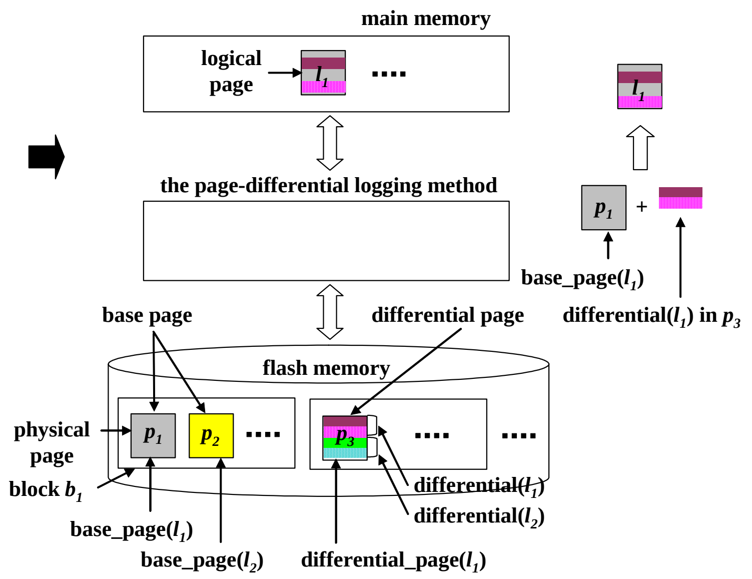
[7] Conceptually, we require an assembly buffer in order to merge the base page with the differential. But, in practice, we can use the logical page itself as the assembly buffer.



(a) The logical pages l_1 and l_2 in memory.



(b) The updated logical pages l_1 and l_2, and the process of writing them into the differential page p_3 in flash memory.



(c) The logical page l_1 recreated from flash memory.

Figure 5. An example of the differential-based approach.

4.2 Data Structures

The data structures used in *flash memory* are base pages, differential pages, and differentials. A base page stores a logical page in its data area and stores the page's type, physical page ID, and creation time stamp in its spare area. Here, the *type* indicates whether the page is a base one or differential one, and the physical page ID represents the unique identifier of a page in the database. The *creation time stamp* indicates when the base page was created.

A differential page stores differentials of logical pages in its data area and stores the page's type in its spare area. A physical page ID and a creation time stamp are stored also in a differential to identify the base page to which the differential belongs and when the differential was created. Therefore, the structure of a differential is in the form of $<$*physical page ID, creation time stamp,* [*offset, length, changed data*]$^{+}>$.

The three data structures used in *memory* are the *physical page mapping table*, the *valid differential count table*, and the *differential write buffer*. The *physical page mapping table* maps a physical page ID into $<$*base page address, differential page address*$>$. This table is used to indirectly reference a base and differential page pair in flash memory because, in flash memory, the positions of the physical pages can be changed by the out-place scheme.

The *valid differential count table* counts the number of valid differentials (i.e., those that have not been obsoleted) in a differential page. When the count becomes 0, the differential page is set to obsolete and made available for garbage collection.

The *differential write buffer* is used to collect differentials of logical pages into memory and later write them into a differential page in flash memory when it is full. The differential write buffer consists of a single page, and thus, the memory usage is negligible. Figure 6 shows the data structures for PDL.

4.3 Algorithms

In this section, we present the algorithms for writing a logical page into flash memory and for recreating a logical page from flash memory. We call them *PDL_Writing* and *PDL_Reading*, respectively.

Figure 7 shows the algorithm *PDL_Writing*. The inputs to the algorithm are the logical page *p* and its physical page ID *pid*. The algorithm consists of the following three steps. In Step 1, we read base_page(*pid*) from flash memory. In Step 2, we create differential(*pid*) by comparing base_page(*pid*)

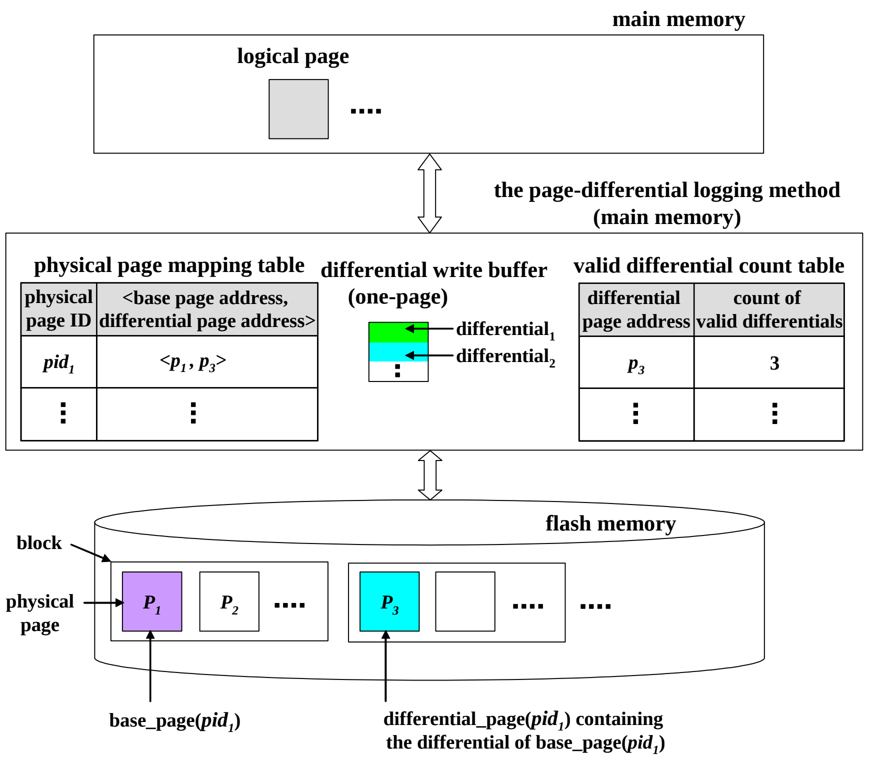

main memory
logical page
....
the page-differential logging method (main memory)
physical page mapping table
physical page ID
<base page address, differential page address>
pid1
<p1, p3>
differential write buffer (one-page)
differential1
differential2
valid differential count table
differential page address
count of valid differentials
p3
3
flash memory
block
physical page
P1
P2
P3
base_page(pid1)
differential_page(pid1) containing the differential of base_page(pid1)

Figure 6. The data structures for PDL.

with p given as an input. In Step 3, we write differential(pid) into the differential write buffer. If old differential(pid) resides in the buffer, we first remove the old one, and then, write the new one. Here, there are three cases according to the size of differential(pid). First, when the size of differential(pid) is equal to or smaller than the free space of the buffer (Case 1), we just write differential(pid) into the buffer. Second, when it is larger than the free space of the buffer but is equal to or smaller than *Max_Differential_Size*[8] (Case 2), we execute the procedure *writingDifferentialWriteBuffer( )* in Figure 8, clear the buffer, and then, write differential(pid) into the buffer. Here, *Max_Differential_Size* is defined as the the maximum size of differentials to be stored in differential pages. The procedure *writingDifferentialWriteBuffer( )* consists of the following two steps. In Step 1, we write the buffer's contents into the differential page q that is newly allocated in flash memory. In Step 2, we update the physical page mapping table *ppmt* and the valid differential count table *vdct*. For each differential d in the buffer, we

[8] In Section 4.1, for ease of exposition, we have explained PDL on the assumption that *Max_Differential_Size* = the size of one physical page. However, in practice, we can adjust it according to the workload. We will show the performance while varying *Max_Differential_Size* later in the experiment section (Section 5).

decrement the count for the old differential page *dp* in *vdct* by executing the procedure *decreaseValidDifferentialCount( )*. Here, if the count becomes 0, we set the differential page to obsolete[9] and make it available for garbage collection. We then set differential_page(*pid_d*) in *ppmt* to the new differential page *q* and increment the count for *q* in *vdct*. Here, *pid_d* is the physical page ID of the base page to which the differential *d* belongs. Third, when it is larger than *Max_Differential_Size* (Case 3), we discard differential(*pid*) and execute the procedure *writingNewBasePage( )* in Figure 8. The procedure consists of the following two steps. In Step 1, we write the logical page *p* itself into the base page *q* that is newly allocated in flash memory. In Step 2, we update *ppmt* and *vdct*. We set the old base page *bp* to obsolete making it available for garbage collection. We then decrement the count for the old differential page *dp* in *vdct* by executing the procedure decreaseValidDifferentialCount( ) and set base_page(*pid*) and differential_page(*pid*) in *ppmt* to *q* and *null*, respectively. Figure 8 shows the procedures for the PDL_Writing algorithm.

```
Algorithm PDL_Writing:
Inputs: (1)  p   /* updated logical page */
        (2) pid   /* physical page ID of p */
Algorithm:
/* Step 1. Reading the base page by looking up the physical page mapping table ppmt */
bp := ppmt(pid).base_page;
Read bp from flash memory;
/* Step 2. Creating a differential */
Create differential(pid) by comparing bp read from flash memory with
        the updated logical page p given as an input;
/* Step 3. Writing the differential into the differential write buffer dwb */
IF old differential(pid) resides in dwb THEN
        Remove old differential(pid);
END /* IF */
IF the size of differential(pid) ≤ free space of dwb THEN                    /* Case 1 */
        Write differential(pid) into dwb;
ELSE IF the size of differential(pid) > free space of dwb AND
          the size of differential(pid) ≤ Max_Differential_Size THEN          /* Case 2 */
        Call writingDifferentialWriteBuffer( );
        Clear dwb;
        Write differential(pid) into dwb;
ELSE IF the size of differential(pid) > Max_Differential_Size THEN           /* Case 3 */
        Discard differential(pid);
        Call writingNewBasePage( );
END /* IF */
```

Figure 7. Writing a logical page into flash memory in PDL.

[9] For the spare area in a page, a write operation that changes a set of bits from 1 to 0 can be repeatedly performed up to four times without an erase operation [6].

```
Procedure writingDifferentialWriteBuffer( ):
Input: dwb   /* differential write buffer */
Algorithm:
/* Step 1. Writing dwb into flash memory as a differential page */
Write its contents into the physical page q that is newly allocated in flash memory;

/* Step 2. Updating the physical page mapping table ppmt and the valid differential count table vdct */
FOR EACH differential d in dwb DO
BEGIN
    pid_d := physical page ID of the base page to which the differential d belongs;
    dp := ppmt(pid_d).differential_page;
    IF dp ≠ null THEN   /* if the differential page already exists */
        Call decreaseValidDifferentialCount(dp);   /* decrement the valid differential count for dp */
    END /* IF */
    ppmt(pid_d).differential_page := q;   /* set the differential page containing d to the new
                                               differential page q */
    vdct(q).count := vdct(q).count + 1;   /* increment the valid differential count for q */
END /* FOR */

Procedure decreaseValidDifferentialCount( ):
Input: dp   /* differential page */
Algorithm:
vdct(dp).count := vdct(dp).count – 1;   /* decrement the valid differential count for dp */
IF vdct(dp).count = 0 THEN
    Set dp to obsolete;
END /* IF */

Procedure writingNewBasePage( ):
Inputs: (1) p   /* logical page */
        (2) pid   /* physical page ID of p */
Algorithm:
/* Step 1. Writing p into flash memory as a new base page */
Write p into the physical page q that is newly allocated in flash memory;

/* Step 2. Updating the physical page mapping table ppmt and the valid differential count table vdct */
bp := ppmt(pid).base_page;
dp := ppmt(pid).differential_page;
Set bp to obsolete;
IF dp ≠ null THEN
    Call decreaseValidDifferentialCount(dp);   /* decrement the valid differential count for dp */
END /* IF */
ppmt(pid).base_page := q;   /* set the base page for the logical page p to the new base page q */
ppmt(pid).differential_page := null;   /* set the differential page for p to null */
```

Figure 8. The procedures for the PDL_Writing algorithm in Figure 7.

Figure 9 shows the algorithm *PDL_Reading*. The input to PDL_Reading is the physical page ID *pid* of the logical page to read. The algorithm consists of the following three steps. In Step 1, we read base_page(*pid*) from flash memory. In Step 2, we find differential(*pid*) of the base_page(*pid*). Here, there are two cases depending on the place where the differential(pid) resides. First, when the differential(pid) resides in the differential write buffer, i.e., when the buffer has not been yet written out to flash memory, we find it from the buffer. Second, when we cannot find it from the buffer, we read differential_page(pid) from flash memory, finding differential(pid) from it. In Step 3, we recreate a logical page p by merging base_page(pid) read in Step 1 with differential(pid) found in Step 2.

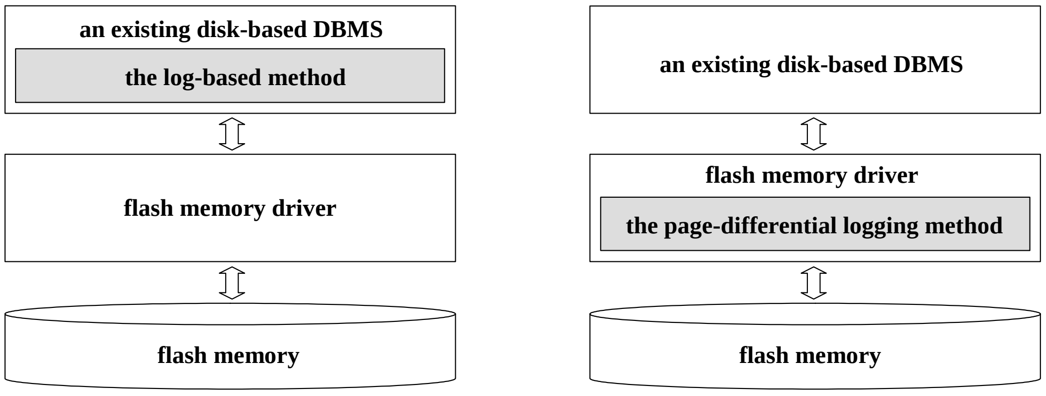
```
Algorithm PDL_Reading
Input: pid   /* physical page ID */
Output: p   /* logical page */
Algorithm:
/* Step 1. Reading the base page  by looking up the physical page mapping table ppmt */
bp := ppmt(pid).base_page;
Read bp from flash memory;

/* Step 2. Finding the differential */
IF differential(pid) resides in the differential write buffer THEN
    Find differential(pid) from the buffer;
ELSE
    dp := ppmt(pid).differential_page;
    IF $dp \neq null$ THEN
        Read dp from flash memory;
        Find differential(pid) from dp read from flash memory;
    ELSE
        Return bp as the result p;   /* there is no differential page */
    END /* IF */
END /* IF */

/* Step 3. Merging the base page with the differential */
Merge bp with differential(pid) to make p;
Return p;
```

Figure 9. Recreating a logical page from flash memory in PDL.

4.4 Discussions

PDL has the following four advantages. (1) As compared with the page-based methods, it has good write performance, i.e., it requires fewer write operations, when we need to reflect an updated logical page into flash memory. This is due to the writing-difference-only principle. (2) As compared with the log-based methods, it has good write performance when a logical page is updated multiple times. This is due to the at-most-one-page writing principle. (3) As compared with the log-based methods, it has good read performance when recreating a logical page from flash memory. This is due to the at-most-two-page reading principle. (4) Moreover, it allows existing disk-based DBMSs to be reused without modification as flash-based DBMSs because it is DBMS-independent.

Figure 10 shows the DBMS architecture that uses flash memory as a secondary storage. The log-based methods need to modify the storage management module of the DBMS so as to write the update log whenever the page is updated as shown in Figure 10 (a). On the other hand, PDL does not need to modify the DBMS but to modify only the flash memory driver[10] because it computes the differential by comparing the whole updated logical page with its base page. Thus, it can be implemented inside the flash memory driver as shown in Figure 10 (b) without affecting the storage manager of the existing DBMS.


an existing disk-based DBMS
the log-based method
flash memory driver
flash memory
an existing disk-based DBMS
flash memory driver
the page-differential logging method
flash memory

(a) The log-based methods. (b) page-differential logging.

Figure 10. The DBMS architecture that uses flash memory as a secondary storage.

PDL, however, has the following minor drawbacks. First, when recreating a logical page from flash memory, PDL has to read one more page than page-based methods do. However, this drawback is relatively minor because the speed of read operation is much faster than that of write or erase operations. Furthermore, if a database is used for read-only access, *PDL reads only one physical page* just like page-based methods since a differential page does not exist (i.e., the base page has not been updated). Thus, in this case, the read performance of PDL is as good as that of the page-based methods. Second, the data size written into flash memory in PDL could be larger than that in log-based methods. It is because the differential contains all the difference between an updated logical page and its base

[10]This flash memory driver corresponds to the FTL shown in Figure 2.

page, while the update log in the log-based methods contains only the difference between an updated logical page and its immediate previous version. However, in spite of this drawback, PDL improves the overall performance significantly because the advantages outweigh these drawbacks. We will show the performance advantages later in the experiment section (Section 5). Table 2 summarizes the differences between PDL and the log-based ones.

Table 2. Comparison of PDL with log-based and page-based ones.

	PDL	log-based methods	page-based methods
data to be written into flash memory	differential	an update log (changed parts only)	the whole page (changed and unchanged parts)
time for writing data into *the write buffer*	only when a logical page needs to be reflected into flash memory	whenever a page is updated	no write buffer
time for writing data into *flash memory*	when the write buffer is full		when a page needs to be reflected into flash memory
number of physical pages to read when recreating a logical page	maximum two pages ($1 \leq n \leq 2$)	multiple pages	one page
architecture	loosely-coupled (DBMS-independent)	tightly-coupled (DBMS-dependent)	loosely-coupled (DBMS-independent)

4.5 Crash Recovery

A storage device with a cache normally supports a *write-through* command that flushes the data written into the cache immediately out to the device. When the write-through command is called, PDL flushes the differential write buffer out into flash memory. In flash memory, the page writing is guaranteed to be atomic at the chip level [9].

When a system failure occurs, we lose the physical page mapping table and the valid differential count table in memory. However, by one scan through physical pages in flash memory, we can reconstruct those tables. Here, the tables are recovered to the state in which data were reflected into flash memory by the write-through call or by flushing the differential write buffer. That is, the data retained in the write buffer only but not written out to flash memory are not recovered in the tables. This is analogous to the situation where data retained only in the file buffer but not written out to disk in a disk file system are not recovered after a system failure. Thus, when persistency of data is required, a write-through call must be used.

If a system failure occurs when a base page (or the differential write buffer) is written into flash memory, but the old base page (or the differential page that does not contain any valid differential) has not yet been set to obsolete in Figure 7, the new base page (or differential page) and the old base page (or differential page) might co-exist in flash memory. Thus, to identify the most up-to-date base page (or differential page), we use the creation time stamp stored in a base page and in each differential in a differential page as in Chang et al. [5].

Figure 11 shows the algorithm for reconstructing the physical page mapping table *ppmt* and the valid differential count table *vdct*. For every physical page r in flash memory, we read the spare area of r and update *ppmt* and *vdct* only if r is not obsolete. Here, there are two cases according to the type of r. First, when r is a base page (Case 1), we check whether ts(r) is more recent than ts(bp), where ts(r) is the creation time stamp of r and ts(bp) is that of the base page bp currently in *ppmt*. If so, r must be a more recent base page. Thus, we set base_page(pid) to r and set the old base page bp to obsolete, where pid is the physical page ID of r. We then check whether ts(r) is more recent than ts(dp, differential(pid)), which is the time stamp of differential(pid) in the differential page dp currently in *ppmt*. If so, the differential(pid) must be obsolete since we have a base page r that is more recent. Thus, we set differential_page(pid) to *null* and decrement the count for the old differential page dp by executing the procedure decreaseValidDifferentialCount( ). If ts(r) is not more recent than ts(bp), we set r to obsolete. Second, when r is a differential page (Case 2), we read the data area of r. For each differential d in r, we check whether ts(d) is more recent than both ts(bp) and ts(dp, differential(pid_d)), where ts(d) is the time stamp of d, ts(bp) is that of the base page bp currently in *ppmt*, and ts(dp, differential(pid_d)) is that of differential(pid_d) in the differential page dp currently in *ppmt*. Here, pid_d is the physical page ID of the base page to which the differential d belongs. If so, d must be a more recent differential of bp than differential(pid_d) currently in *ppmt*. Thus, we set differential_page(pid_d) to r, decrement the count for the old differential page dp by executing the procedure decreaseValidDifferentialCount( ), and increment the count for the new differential page r. If r does not contain any valid differential after processing all the differentials in r, we set r to obsolete.

```
Algorithm PDL_RecoveringfromCrash
/* Reconstructing the physical page mapping table ppmt and the valid differential count table vdct */
Initialize ppmt and vdct;
FOR EACH physical page r in flash memory DO
BEGIN
    Read the spare area of r from flash memory;
    IF IS_OBSOLETE_PAGE(r) THEN
        CONTINUE;
    END /* IF */
    IF IS_BASE_PAGE(r) THEN   /* Case 1: r is a base page */
        pid := physical page ID of r;
        bp := ppmt(pid).base_page;
        dp := ppmt(pid).differential_page;
        /* ts(x, y) returns the creation time stamp as follows:
           (1) if x is a base page or a differential, returns the time stamp of x (here, y can be omitted)
           (2) if x is a differential page, returns the time stamp of differential y in x */
        IF ts(r) > ts(bp) THEN   /* r is a more recent base page */
            Set bp to obsolete;
            ppmt(pid).base_page := r;   /* set the base page with pid to the new base page r */
            IF ts(r) > ts(dp, differential(pid)) THEN   /* r is more recent than differential(pid) in dp */
                Call decreaseValidDifferentialCount(dp);   /* decrement the valid differential count for dp */
                ppmt(pid).differential_page := null;   /* set the differential page containing differential(pid) to null */
            END /* IF */
        ELSE   /* bp is a more recent base page */
            Set r to obsolete;
        END /* IF */
    ELSE   /* Case 2: r is a differential page */
        Read the data area of r from flash memory;
        FOR EACH differential d in r DO
        BEGIN
            pid_d := physical page ID of the base page to which the differntial d belongs;
            bp := ppmt(pid_d).base_page;
            dp := ppmt(pid_d).differential_page;
            IF ts(d) > ts(bp) AND ts(d) > ts(dp,differential(pid_d)) THEN   /* d is more recent than bp and differential(pid_d) in dp */
                Call decreaseValidDifferentialCount(dp);   /* decrement the valid differential count for dp */
                ppmt(pid_d).differential_page := r;   /* set the differential page containing d to the new differential page r */
                vdct(r).count := vdct(r).count + 1;   /* increment the valid differential count for r */
            END /* IF */
        END /* FOR */
        IF vdct(r). count = 0 THEN    /* r does not contain any valid differential */
            Set r to obsolete;
        END /* IF */
    END /* IF */
END /* FOR EACH */
```

Figure 11. The algorithm for reconstructing the physical page mapping table and the valid differential count table upon system failure.

In Figure 11, we set two kinds of useless pages to obsolete: (1) base pages that are not recent but have not been set to obsolete and (2) differential pages that do not contain valid differential but have not been set to obsolete. These pages can occur in flash memory when a system failure occurs if a base page (or the differential write buffer) has been written into flash memory, but the old base page (or the differential page that does not contain valid differentials) has not yet been set to obsolete.

The algorithm PDL_RecoveringfromCrash guarantees that recovery is normally performed even when a system failure repeatedly occurs during the process of restarting the system. The reason is that the algorithm does not change data in the flash memory except setting the useless pages (i.e., the pages that are no longer used, but have not been set to obsolete) to obsolete. Setting useless pages to obsolete does not affect the recovery process of reconstructing the physical page mapping table and the valid differential count table.

Since scanning the entire flash memory of 1 Gbytes takes approximately 60 seconds (derived from Table 1 in Section 2), the scan time can be practically accommodated. To recover the physical page mapping table without scanning all the physical pages in flash memory, we have to log the changes in the mapping table into flash memory. We leave this extension as a further study.

We note that we can implement the proposed PDL and recovery techniques in a DBMS that uses flash memory to support transactional database recovery just as we do in a DBMS built on top of an O/S file system by using the write-through facility whenever persistency of a write operation is required (e.g., when writing the 'transaction commit' log record).

5 Performance Evaluation

5.1 Experimental Data and Environment

We compare the data access performance of PDL proposed in this paper with those of the page-based and log-based methods discussed in Section 3. We use the wall clock time taken to access data from flash memory (we call it *the I/O time*) as the measure. Here, as the page-based method, we use the one employing the out-place update (OPU) scheme with the page-level mapping technique, which is known to have good performance even though the method consumes memory excessively [9]. We also compare with the in-place update method (IPU). As the log-based method, we use the in-page logging method (IPL) proposed by Lee and Moon [14].

We use the synthetic relational data of 1 Gbytes and update operations for comparing data access performance of the three methods. We define an *update* operation as consisting of the following three steps: (1) reading the addressed page; (2) changing the data in the page; and (3) writing the updated page. The reading step (1) creates a logical page by reading physical pages from flash memory, and the writing step (3) writes the updated logical page as one or more physical pages into flash memory. The experiments are designed this way to exclude the buffering effect in the DBMS. Therefore, we can measure read, write as well as overall performance by executing only update operations.

The I/O time is affected by $N\_updates\_till\_write$ and $\%ChangedByOneU\_Op$. Here, $N\_updates\_till\_write$ is the number of update operations applied to a logical page in memory from the time it is recreated from flash memory until the time it is reflected back into flash memory, $\%ChangedByOneU\_Op$ is the percentage of data changed in a logical page by a single update operation. Here, the portion of data to be changed is randomly selected. We also compare the performance of various mixes of read-only and update operations varying the percentage of the update operations ($\%UpdateOps$). Besides, we measure the performance as we vary the performance parameters of flash memory (i.e., the I/O times for read and write operations in Table 1). We also compare the longevity of flash memory. Finally, we perform the TPC-C benchmark [20] as a real workload. Table 3 summarizes the experiments and parameters.

In each experiment, garbage collection is invoked whenever there is no more free page in flash memory [11]. Here, the cost (time) of garbage collection is amortized into that of the write operation because garbage collection is incurred by the accumulated effect of write operations. We repeatedly execute experiments so that garbage collection is invoked for each block at least ten times on the average after loading the database in order to make the database to reach a steady state.

[11] In IPL, garbage collection is invoked during the process of merging.

Table 3. Experiments and parameters.

Experiments		Parameters	
Exp. 1	Read, write, and overall time per update operation	$\%ChangedByOneU\_Op$	2
		$N\_updates\_till\_write$	1
Exp. 2	Overall time per update operation as $N\_updates\_till\_write$ is varied	$\%ChangedByOneU\_Op$	2
		$N\_updates\_till\_write$	$1 \sim 8$
Exp. 3	Overall time per update operation as $\%ChangedByOneU\_Op$ is varied	$\%ChangedByOneU\_Op$	$0.1 \sim 100$
		$N\_updates\_till\_write$	1, 5
Exp. 4	Overall time per operation for the mixes of read-only and update operations as $\%UpdateOps$ is varied	$\%ChangedByOneU\_Op$	2
		$N\_updates\_till\_write$	1, 5
		$\%UpdateOps$	$0 \sim 100$
Exp. 5	Overall time per update operation as the parameters of flash memory are varied	$\%ChangedByOneU\_Op$	2
		$N\_updates\_till\_write$	1
		T_{read}	$10 \sim 1500$
		T_{write}	500, 1000
Exp. 6	Number of erase operations per update operation as $N\_updates\_till\_write$ is varied	$\%ChangedByOneU\_Op$	2
		$N\_updates\_till\_write$	$1 \sim 8$
Exp. 7	I/O time per transaction for TPC-C data as the DBMS buffer size is varied	DBMS buffer size	$1 \sim 100$ Mbytes ($0.1 \sim 10$ % of database size)

For the experiments, we have implemented an emulator of a 2-Gbyte flash memory chip using the parameters shown in Table 1 [12]. We also have implemented the four methods: PDL (x), OPU, IPU, and IPL (y) [13] [14] for PDL and OPU. Here, x is *Max_Differential_Size* (defined in Section 4.3 in p. 17), and y is the amount of log pages in each block. We used the Odysseus ORDBMS [23, 24] as the storage system. Here, PDL, OPU, and IPU are implemented outside the DBMS, and IPL inside the DBMS. We conducted all experiments on a Pentium 4 3.0 GHz Linux PC with 2 Gbytes of main memory. We set the size of a logical page to be 2 Kbytes, which is the size of a physical page in flash memory. We also test the case with a logical page of 8 Kbytes as was done by Lee and Moon [14].

[12] For each operation, the emulator returns the required time in the flash memory, which is specified in Table 1, while writing and reading the data to and from the disk. The data are in exactly the same format in disk as would be stored in flash memory. Thus, access time using the emulator must be identical to that using the real flash memory.

[13] We set the size of log buffer for each logical page to the size of a logical page $\times \frac{1}{16}$ as was used by Lee and Moon [14].

[14] We do not use wear-leveling in this paper, but the same wear-leveling techniques can be applied to these methods. We use the same garbage collection method suggested by Woodhouse [21]

5.2 Results of the Experiments

Experiment 1:

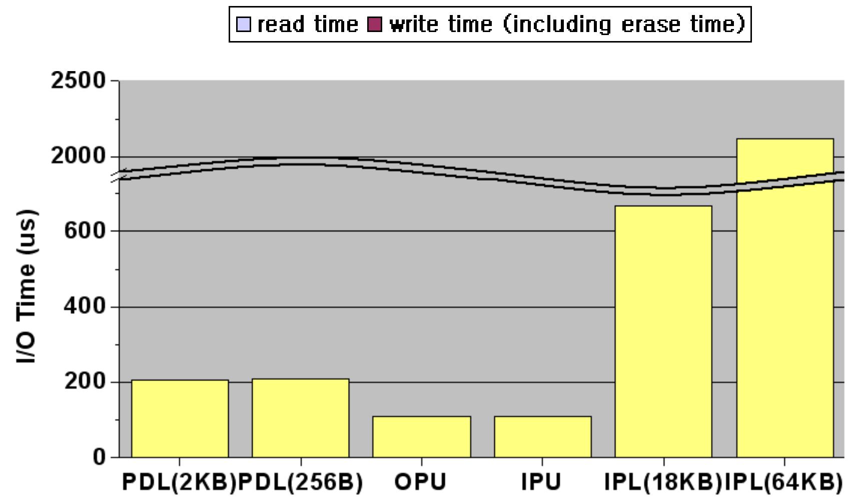
Figure 12 shows the read, write, and overall time per update operation for the six methods: IPL (18KB), IPL (64KB), PDL (2KB), PDL (256B), OPU, and IPU. For IPL (y), we have varied y from 8 Kbytes to 64 Kbytes. Among them, we select IPL (18KB) and IPL (64KB) because they have the best and worst overall time for update operations, respectively. For PDL, we select PDL (2KB) and PDL (256B) because the amounts of differential pages in them are similar to those of log pages in IPL (64KB) and IPL (18KB), respectively. Specifically, IPL (64KB) and PDL (2KB) use 50 % of flash memory for storing log/differential pages. IPL (18KB) and PDL (256B) use 14.1 % and 11.1 % of flash memory, respectively.

Figure 12 (a) shows that the I/O time of the reading step per update operation is in the following order: IPL (64KB), IPL (18KB), PDL (2KB) / PDL(256B), and OPU / IPU. This result is consistent with what was discussed in Sections 3 and 4. OPU and IPU require one read operation. PDL requires at most twice as many read operations. IPL requires multiple read operations. We note that, when we perform read-only operations, we can also achieve the same result as is shown in Figure 12 (a).

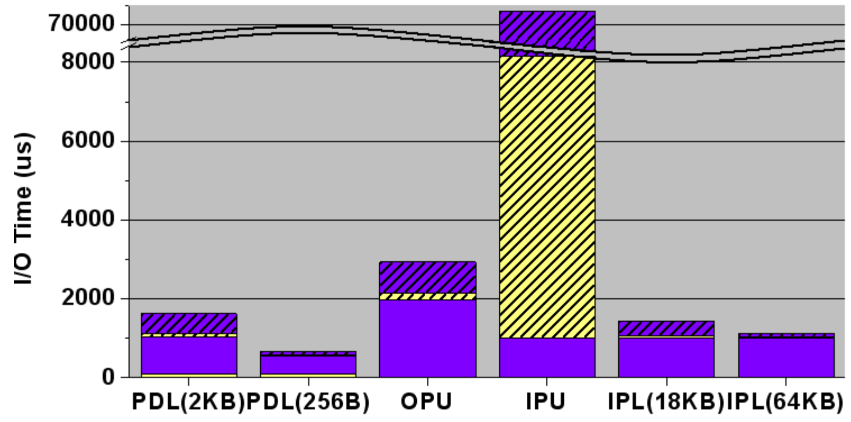
Figure 12 (b) shows that the I/O time of the writing step is in the following order: IPU, OPU, PDL (2KB), IPL (18KB), IPL (64KB), and PDL (256B). Here, the slashed area indicates the I/O time for garbage collection. The result is also consistent with the discussions in Sections 3 and 4. For an update operation, OPU requires two write operations: one for writing the updated page into flash memory and another for setting the original page to obsolete. However, IPL requires only one write operation for writing the log buffer into flash memory. PDL (2KB) requires two write operations approximately for every two update operations: one for writing the differential write buffer into flash memory and another for setting one (on the average) differential page to obsolete[15] because the size of a differential is approximately half a page on the average[16]. Thus, PDL (2KB) requires approximately one write operation for an update operation on the average. PDL (256B) requires a less number of write operations than PDL (2KB) does since the differential write buffer is filled less frequently. But, PDL additionally

[15] When the count of valid differentials in *vdct* becomes 0, we set the differential page to obsolete.

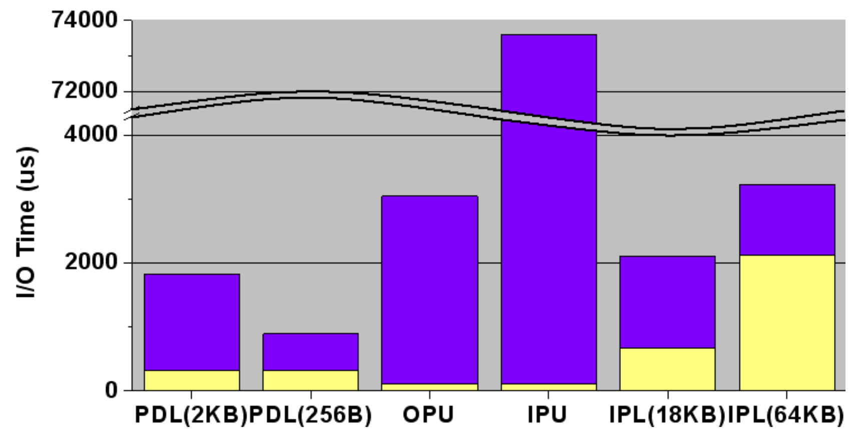
[16] Since the size of a differential changes from 0 to 1 page size and back to 0 (Case 3 in Figure 7) as updating a logical page is repeated, the size of a differential in a steady state is approximately half a page on the average.



(a) The I/O time of the reading step.



(b) The I/O time of the writing step. Slashed parts indicate the time for garbage collection. Lighter areas represent read time.



(c) The overall time per update operation including read and write times in (a) and (b).

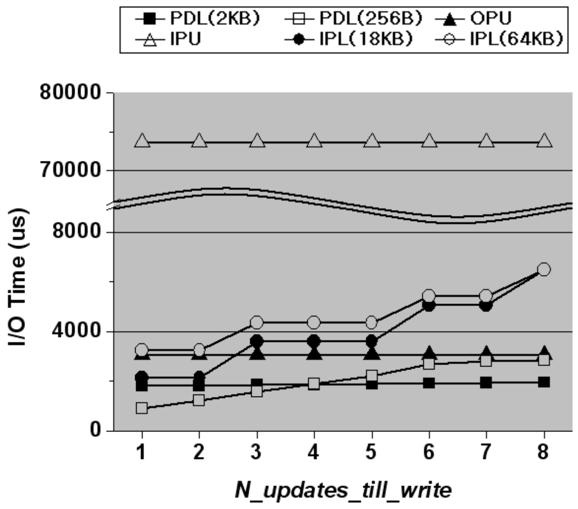
Figure 12. The read, write, and overall time per update operation ($N\_updates\_till\_write = 1$, $\%ChangedByOneU\_Op = 2$, database size = 1 Gbytes, $T_{read} = 110\,\mu s, T_{write} = 1010\,\mu s$).

requires one read operation for reading the base page in from flash memory in order to create the differential. Here, each method includes a certain amount of read cost, which is incurred by garbage collection and amortized into the write cost. We note that PDL (256B) outperforms the other methods due to less frequent writing of the differential write buffer.

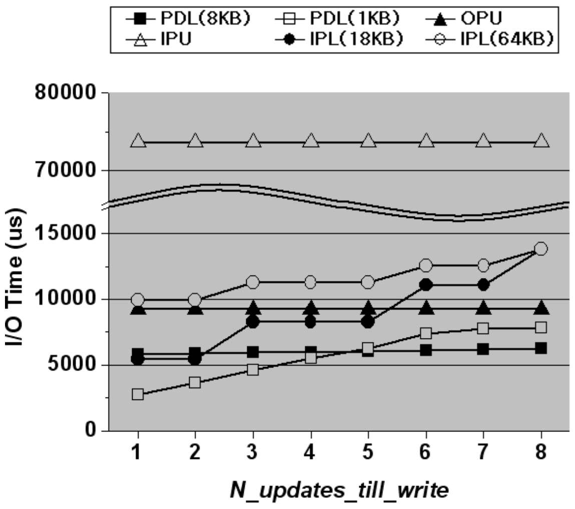
Figure 12 (c) shows the overall time per update operation combining the I/O times shown in Figures 12 (a) and (b). PDL (256B) has good read and write performance as shown in Figures 12 (a) and (b), and thus, has the best overall time for an update operation. (This corresponds to Figure 13 (a) when $N\_updates\_till\_write = 1$.)

Experiment 2:

Figure 13 shows the overall time per update operation of the six methods as *N_updates_till_write* is varied. First, the I/O time of OPU and IPU is steady regardless of the parameter because they always write the whole page when reflecting an updated logical page into flash memory. Next, the I/O time of IPL increases in a stepwise manner. The reason for this behavior is that the number of write operations for a logical page is computed as $\lceil \frac{the\ size\ of\ update\ logs}{the\ size\ of\ log\ buffer} \rceil$. Here, the size of the update logs to be written increases linearly as *N_updates_till_write* increases because IPL keeps all the update logs of a logical page. (We note that this process of writing is not bound by merging while the reading process is.) Finally, the I/O time of PDL (2KB) increases only very slightly as *N_updates_till_write* increases because the size of the overlap among the changed parts becomes larger as *N_updates_till_write* increases with the total size of the difference being limited to one page. The I/O time of PDL (256B) increases approximately linearly as *N_updates_till_write* increases because the size of the overlap is small. As *N_updates_till_write* increases, the I/O time of PDL (256B) approaches that of OPU because the logical page itself (rather than the differential) is written into flash memory as the size of the differential becomes larger than *Max_Differential_Size* (Case 3 in Figure 7). As a result, PDL (256B) outperforms OPU, IPU, and IPL. The result when the size of a logical page is 8 Kbytes shows a tendency similar to that when the size of a logical page is 2 Kbytes.



(a) size of a logical page = 2 Kbytes.



(b) size of a logical page = 8 Kbytes.

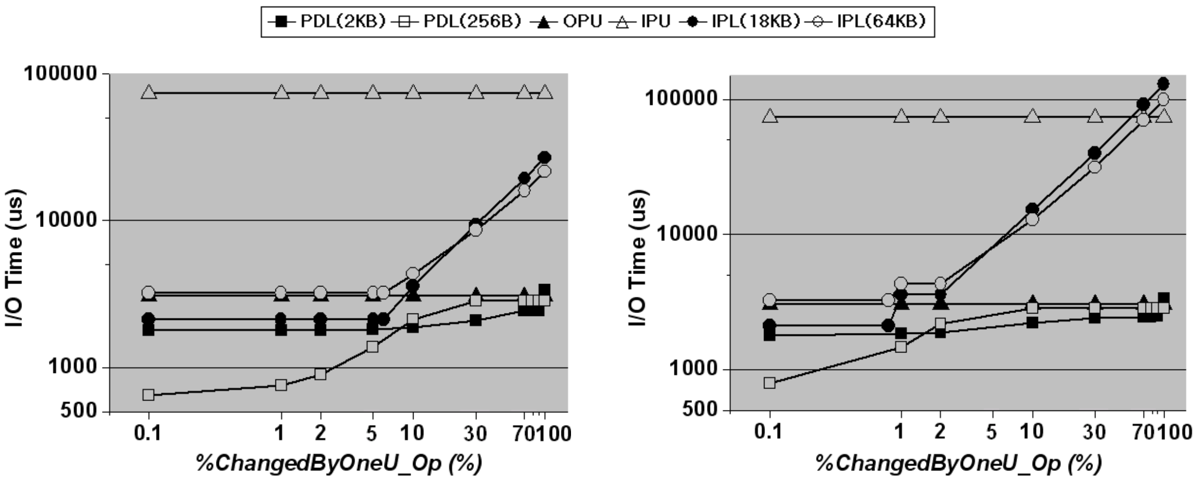
Figure 13. The overall time per update operation as *N_updates_till_write* is varied ($\%ChangedByOneU\_Op = 2$).

Experiment 3:

Figure 14 shows the overall time per update operation for the six methods as $\%ChangedByOneU\_Op$ is varied. The result is consistent with what we observed in Figure 13. We note that PDL (256B) outperforms OPU, IPU, and IPL for the same reason as in Figure 13. When $\%ChangedByOneU\_Op \approx 100$, the I/O time of PDL (2KB) is slightly larger than that of OPU because, while the two methods require the same number of write operations, PDL (2KB) needs three times as many read operations—for reading the base page and the differential page when recreating a logical page from flash memory, and then, for reading the base page again to create the differential when reflecting the updated logical page into flash memory.

Experiment 4:

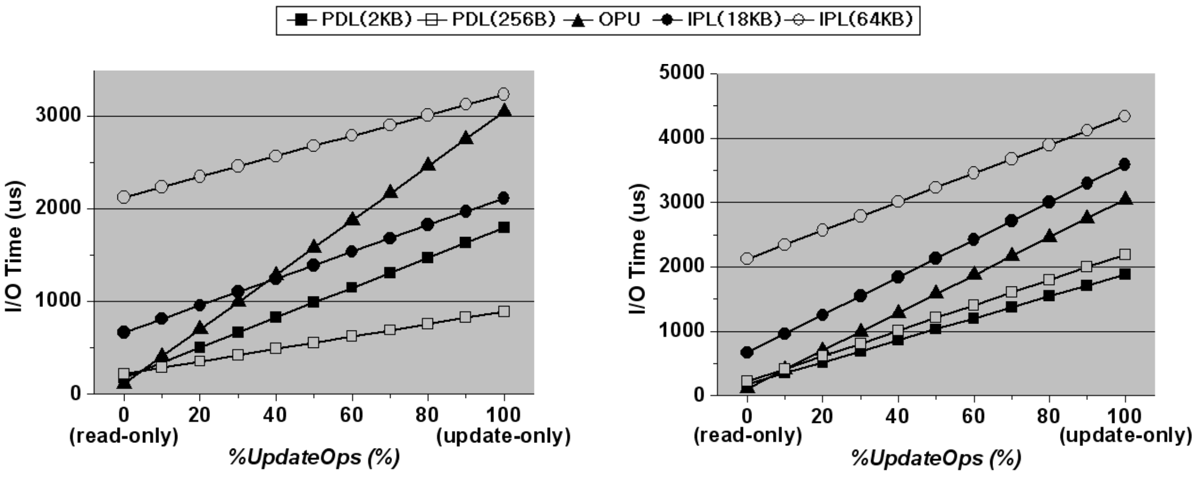
Figure 15 shows the results of Experiment 4. When updates are rare (i.e., $\%UpdateOps \approx 0$), OPU outperforms PDL and IPL (see Figure 12 (a)). As $\%UpdateOps$ increases, PDL becomes superior to OPU because of its superiority in update performance (see Figure 12 (c)). We also note that PDL always outperforms IPL. In summary, for various mixes of read-only and update operations, PDL (256B)



(a) $N\_updates\_till\_write = 1$. (b) $N\_updates\_till\_write = 5$.

Figure 14. The overall time per update operation as $\%ChangedByOneU\_Op$ is varied ($N\_updates\_till\_write = 1, 5$).

improves performance by 0.5 ∼ 3.4 times over OPU and by 1.6 ∼ 3.1 times over IPL (18KB) and by 2.0 ∼ 9.7 times over IPL (64KB). We note that the case of 0.5 times over OPU is the special case where all transactions are read-only (i.e., $\%UpdateOps = 0$).

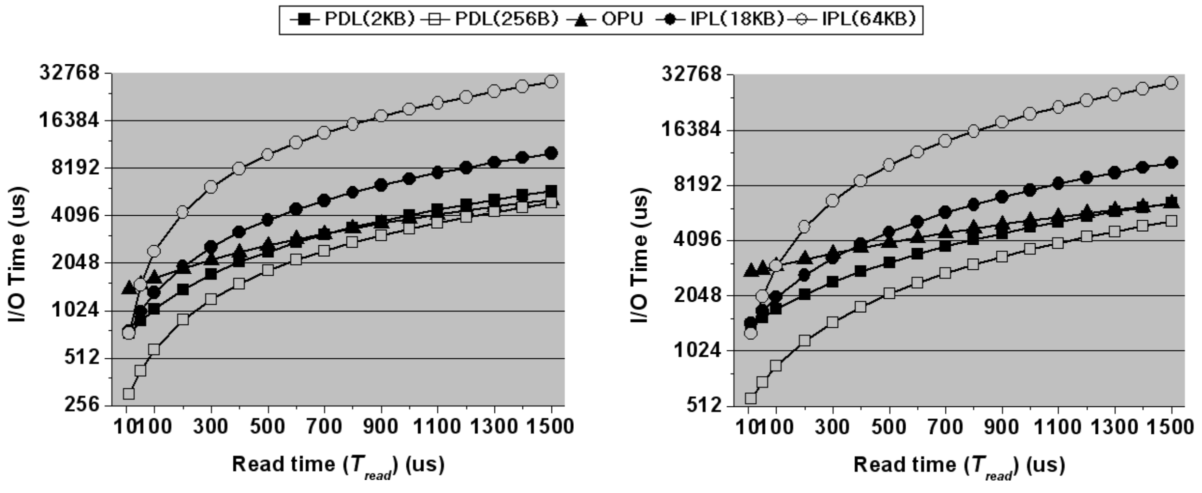


(a) $N\_updates\_till\_write = 1$. (b) $N\_updates\_till\_write = 5$.

Figure 15. The overall time per operation for the mixes of read-only and update operations as $\%UpdateOps$ is varied ($\%ChangedByOneU\_Op = 2$).

Experiment 5:

Figure 16 shows the overall time per update operation as the T_{read} and T_{write} parameters of flash memory are varied. We observe that PDL (256B) always outperforms OPU and IPL. As the read time (T_{read}) increases, OPU becomes superior to PDL (2KB) or IPL. We have this result because OPU has superiority in read performance (see Figure 12 (a)). We note that PDL (256B) outperforms OPU and IPL regardless of the T_{read} and T_{write} parameters of flash memory.



(a) $T_{write} = 500\,\mu s$. (b) $T_{write} = 1000\,\mu s$.

Figure 16. The overall time per update operation as the T_{read} and T_{write} parameters of flash memory are varied ($N\_updates\_till\_write = 1, \%ChangedByOneU\_Op = 2, T_{erase} = 1500\,\mu s$).

Experiment 6:

Figure 17 shows the number of erase operations per update operation as $N\_updates\_till\_write$ is varied. We observe that, when $N\_updates\_till\_write = 1$, the number of erase operations per update operation is in the following order: OPU, PDL (2KB), IPL (18KB), PDL (256B), and IPL (64KB). Thus, IPL (64KB) has the best longevity among the five methods. But, it has poor performance for the mixes of read-only and update operations as shown in Figure 15. PDL (256B) has good longevity next to IPL (64KB). Besides, it has significantly good performance for the mixes of read-only and update operations.

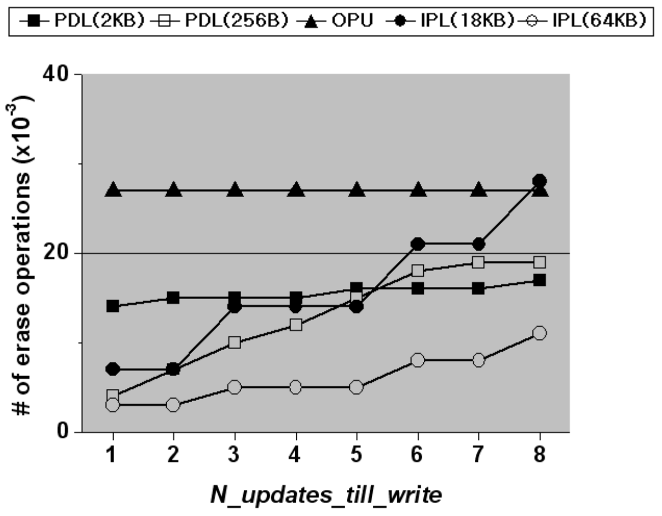


Figure 17. The number of erase operations per update operation as *N_updates_till_write* is varied (*%ChangedByOneU_Op* = 2).

Experiment 7:

Figure 18 shows the results of the TPC-C benchmark. We observe that the I/O time is in the following order: IPL (64KB), IPL (18KB), OPU, PDL (2KB), and PDL (256B). The result shows that PDL outperforms other methods in real workloads as well.

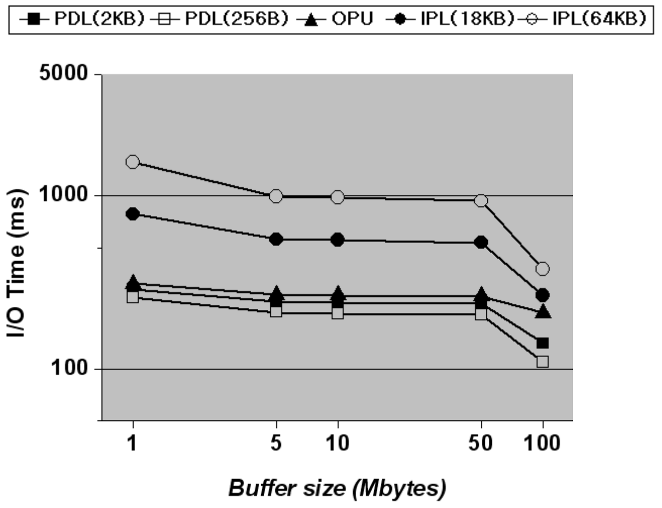


Figure 18. TPC-C benchmark: I/O time per transaction as the DBMS buffer size is varied (database size = 1 Gbytes).

6 Conclusions

We have proposed a novel approach for storing data called *page-differential logging* for flash-based storage systems. We have defined the notion of the differential and presented the algorithms for reading and writing pages into flash memory using the differential.

We have identified three design principles: writing-difference-only, at-most-one-page writing, and at-most-two-page reading. These principles guarantee good performance for both read and write operations. We have shown that our method conforms to these principles.

Page-differential logging is DBMS-independent, i.e., it allows existing disk-based DBMSs to be reused as flash-based DBMSs just by modifying the flash memory driver. In addition, it improves the longevity of flash memory by reducing the number of erase operations compared with existing page-based methods.

We have performed extensive experiments to compare the performance of page-differential logging with existing page-update methods. Through these experiments, we have shown that the performance of our method is superior to those of page-based and log-based methods — except when all transactions are read-only on already updated pages. We also performed experiments as the performance figures of read and write operations change. The results show that our method (in particular, PDL (256B)) is always superior to other methods. Thus, the results indicate that page-differential logging can be the preferred technique for commercial products [17]. We also performed experiments to compare various methods for the longevity of flash memory. The results show that our method (in particular, PDL (256B)) improves the longevity of flash memory compared with OPU and IPL (18KB). Finally, we performed the TPC-C benchmark as the DBMS buffer size is varied. The results show that our method (in particular, PDL (256B)) outperforms other methods by 1.2 $\sim$ 6.1 times. This shows effectiveness of our method under real workloads.

Currently, we are implementing page-differential logging on a flash memory embedded board. Such an augmented board is to be incorporated to our Odysseus DBMS [23, 24]. The resulting system will

[17] Commercial SSD's offer average write time comparable to read time by exploiting parallelism, but individual NAND flash chips typically have asymmetric read/write times.

facilitate various flash-memory-dependent optimizations in various components of the DBMS such as the indexes, buffer, sort module, and query optimizer. We also note that, due to its DBMS-independent nature, page-differential logging can be employed by the manufacturer in the FTL of commercial SSD's. We leave these issues as future work.

7 Acknowledgement

This research was partially supported by the National Research Lab Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (No. 2009-0083120). Page-differential logging has been introduced as a part of Yi-Reun Kim's Ph.D. dissertation [7] and patented in Korea [22] in Nov. 2009.

References

[1] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R., "Design Tradeoffs for SSD performance," In *Proc. USENIX Annual Technical Conference,* pp. 57–70, Boston, Massachusetts, June 2008.

[2] Aleph One Ltd., Yet Another Flash File System (YAFFS), 2002. Available at http://www.yaffs.net

[3] Ban, A., Flash File System, US patent 5404485, 1995.

[4] Chang, L. and Kuo, T., "An Efficient Management Scheme for Large-Scale Flash Memory Storage Systems," In *ACM Symposium on Applied Computing (SAC)*, pp. 862–868, Nicosia, Cyprus, Mar. 2004.

[5] Chang, L. and Kuo, T., "Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation," In *ACM Transactions on Storage*, pp. 381–418, Vol. 1, No. 4, 2005.

[6] Gal, E. and Toledo, S., "Algorithms and Data Structures for Flash Memories," In *ACM Computing Surveys*, pp. 138–163, Vol. 37, No. 2, 2005.

[7] Kim, Y., Managing MEMS and Flash Storage Devices in a DBMS, Ph.D. Dissertation, Dept. of Computer Science, KAIST, Korea, Aug. 2009.

[8] Kim, G., Baek, S., Lee, H., Lee, H., and Joe, M., "LGeDBMS: a Small DBMS for Embedded System with Flash Memory," In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 1255–1258, Seoul, Korea, Sept. 2006.

[9] Kim, J., Kim, J. M., Choi, J., Jeong, J., Noh, S., and Min, S.,"Design and Implementation of a Flash Memory Based Storage System for Mobile Devices," Technical Report (SNU-CE-TR-2001-1), School of Computer Science and Engineering, Seoul National University, Korea, Apr. 2001.

[10] Kawaguchi, A., Nishioka, S., and Motoda, H., "A Flash-Memory Based File System," In *Proc. USENIX Annual Technical Conference*, pp. 155–164, New Orleans, Louisiana, Jan. 1995.

[11] Kim, T., Shin, K., Lee, T., and Chung, K.,"Design of a Reliable NAND Flash Software for Mobile Device," In *Proc. 9th Int'l Conf. on Computer and Information Technology (CIT)*, pp. 173–174, Bhubaneswar, India, Dec. 2006.

[12] Koltsidas, I. and Viglas, S. D., "Flashing Up the Storage Layer," In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 514–525, Auckland, New Zealand, Aug. 2008.

[13] Lee, S., Choi, W., Park, D., "FAST: An Efficient Flash Translation Layer for Flash Memory," In *Proc. 1st Int'l Workshop on Embedded Software Optimization (ESO)*, pp. 879–887, Seoul, Korea, Aug. 2006.

[14] Lee, S. and Moon, B., "Design of Flash-Based DBMS: An In-Page Logging Approach," In *Proc. Int'l Conf. on Management of Data*, pp. 55–66, ACM SIGMOD, Beijing, China, June 2007.

[15] Nath, S. and Gibbons, P. B., "Online Maintenance of Very Large Random Samples on Flash Storage," In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 970–983, Auckland, New Zealand, Aug. 2008.

[16] Park, C., Seo, J., Seo, D., Kim, S., and Kim, B., "Cost-Efficient Memory Architecture Design

of NAND Flash Memory Embedded Systems," In *Proc. Int'l Conf. on Computer Design (ICCD)*, pp. 474–480, San Jose, California, Oct. 2003.

[17] Rosenblum, M. and Ousterhout, J. K., "The Design and Implementation of a Log-Structured File System," In *ACM Transactions on Computer Systems (TOCS)*, pp. 26–52, Vol. 10, No. 1, 1992.

[18] Samsung Electronics, Datasheet: 1 G x 8 Bit / 2 G x 8 Bit / 4 G x 8 Bit NAND Flash Memory, 2005. Available at http://www.DataSheet4U.com

[19] Samsung Semiconductor, Flash SSD, 2008. Available at http://www.samsung.com/global/business/semiconductor/products/flash/Products_FlashSSD.html

[20] Transaction Processing Performance Council (TPC), TPC Benchmark C. Standard Specification, 2002. Available at http://www.tpc.org/tpcc

[21] Woodhouse, D., "JFFS: The Journaling Flash File System," In *Proc. Ottawa Linux Symposium*, Ottawa, Canada, July 2001.

[22] Whang, K and Kim, Y., A Method to Store Data into Flash Memory in a DBMS-independent Manner Using the Page-Differential, Korean Patent 10-0929371, Nov. 24, 2009.

[23] Whang, K., Lee, M., Lee, J., Kim, M., and Han, W., "Odysseus: a High-Performance ORDBMS Tightly-Coupled with IR Features," In *Proc. 21st IEEE Int'l Conf. on Data Engineering (ICDE)*, Tokyo, Japan, pp. 1104–1105, Apr. 2005. This paper received the best demonstration award.

[24] Whang, K., Lee, J., Kim, M., Lee, M., and Lee, K., "Odysseus: a High-Performance ORDBMS Tightly-Coupled with Spatial Database Features," In *Proc. 23rd IEEE Int'l Conf. on Data Engineering (ICDE)*, pp. 1493–1494, Istanbul, Turkey, Apr. 2007.

[25] Wu, C., Kuo, T., and Chang, L., "An Efficient B-Tree Layer for Flash-Memory Storage Systems," In *ACM Transactions on Embedded Computing Systems*, Vol. 6, No. 3, 2007.